Massively parallel Monte Carlo for many-particle simulations on GPUs

Joshua A. Anderson,[†] Eric Jankowski,[†] Thomas L. Grubb,[‡] Michael Engel,[†] and Sharon C. Glotzer[*,†,‡]

Department of Chemical Engineering, University of Michigan, Ann Arbor, MI 48109, USA, and Department of Materials Science and Engineering, University of Michigan, Ann Arbor, MI 48109, USA

E-mail: sglotzer@umich.edu

Abstract

Current trends in parallel processors call for the design of efficient massively parallel algorithms for scientific computing. Parallel algorithms for Monte Carlo simulations of thermodynamic ensembles of particles have received little attention because of the inherent serial nature of the statistical sampling. In this paper, we present a massively parallel method that obeys detailed balance and implement it for a system of hard disks on the GPU. We reproduce results of serial high-precision Monte Carlo runs to verify the method. This is a good test case because the hard disk equation of state over the range where the liquid transforms into the solid is particularly sensitive to small deviations away from the balance conditions. On a GeForce GTX 680, our GPU implementation executes 95 times faster than on a single Intel Xeon E5540 CPU core, enabling 17 times better performance per dollar and cutting energy usage by a factor of 10.

1 Introduction

During the last decades computational scientists have enjoyed a doubling of performance for single-threaded applications every two years solely from improvements in computer architecture. This is no longer the case. Current processor designs run into the Power Wall, which limits attainable clock speeds in a given power budget, and the Instruction Level Parallelism (ILP) Wall, which exists because there is only so much ILP that can be extracted from a typical program.[1] Moore's law still holds, for now, and the additional transistors go into increasing the core counts on each new chip. Consequently, researchers must utilize parallelism to execute larger, longer, or more demanding calculations and simulations.

As of this publication, a cluster of networked multi-core CPUs is the most common system architecture. In an alternative approach, a single graphics processing unit (GPU) can execute thousands of instructions at the same time and provides the performance of a small cluster at a fraction of the cost.[2] GPUs are becoming popular as desktop 'personal supercomputers' and as coprocessors in heterogeneous clusters. A successful GPU algorithm divides a given computation into a maximal number of identical, fully independent, and simple tasks, called "threads". To fully utilize their potential it is necessary to design not just parallel, but *massively parallel* algorithms that scale to thousands of threads. Over the last few years, many problems have successfully been adapted to GPUs. One example is the molecular dynamics (MD) method for simulating thermodynamic ensembles of particles, which is well suited for mas-

[*]To whom correspondence should be addressed

[†]Department of Chemical Engineering, University of Michigan, Ann Arbor, MI 48109, USA

[‡]Department of Materials Science and Engineering, University of Michigan, Ann Arbor, MI 48109, USA

sive parallelization. Numerous MD software packages support GPUs, including HOOMD-blue,[3,4] LAMMPS,[5] AMBER,[6,7] NAMD,[8] OpenMM,[9] FEN ZI,[10] HALMD,[11] and the work by Rapaport.[12]

A case where implementation to GPUs has so far not been achieved is Monte Carlo (MC) applied to many-particle systems. MC is a statistical, rather than deterministic, sampling method that, appropriately implemented, samples the microstates of desired thermodynamic ensembles. It is the method of choice in many situations because it only requires an interaction potential, not a force field, allowing *e.g.* the use of non-differentiable pair potentials. Such potentials are useful in the simulation of hard particles, which interact solely via excluded volume. MC is also flexible in the sense that a wide variety of update moves can be applied.[13–16] MC is easy to implement on serial machines where each step selects a new microstate at random and accepts or rejects the new microstate based on the Boltzmann factor. An example trial move involves translating a single particle in a random direction. Since the acceptance of a trial move depends on results of prior moves, subsequent moves usually cannot be performed independently. A massively parallel algorithm must not only update a large number of particles at the same time, but also do so in a statistically correct way. This can be achieved by obeying detailed balance, which is not a trivial task. For these reasons, parallel MC codes for particle systems have received much less attention in the literature than parallel MD.

Available parallel MC algorithms fall into several categories. Most of them rely on domain decomposition schemes to update portions of the problem in parallel. Lattice MC has been employed for Ising models[17,18] including GPU implementations.[19,20] Related schemes for particle systems exist,[21–24] but none scale up to thousands of threads. Moreover, the updating of domains employed in some of those methods can introduce a sampling bias that precludes the balance conditions. Asynchronous parallel algorithms[25,26] are poorly suited for GPUs because of their extensive inter-thread communication. Hybrid approaches that employ MD trajectories to create trial configurations[27,28] can be an effective way to exploit GPU parallelism, but require substantial additional programming and are not guaranteed to evolve any faster than MD alone.

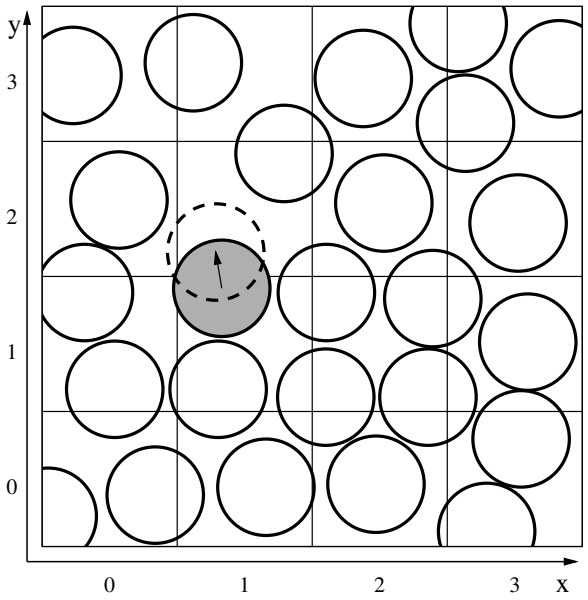

Figure 1: In a serial hard disk simulation, each trial move typically consists of the following: (i) Select a single disk at random, (ii) apply a random displacement to it, and (iii) accept the move if it generates no overlaps. The cell list enables $O(1)$ overlap checks by limiting the search space to only nine local cells. A sweep is defined as N consecutive trial moves, where N is the number of disks in the simulation.

In this paper, we develop an algorithm for massively parallel Monte Carlo (MPMC) simulations of many-particle systems that obeys detailed balance. As a test case, we implement it on the GPU for a system of hard disks in two dimensions (Figure 1) and validate it with comparisons to recent large-scale serial event-chain MC simulations.[29] Our algorithm is not specific to this system, and it is valid for any MC simulation with local interactions between particles or lattice sites on any massively parallel computer architecture.

2 Algorithm

In developing any parallel application, the programmer must identify the computations that can be executed simultaneously. As the total work is broken into smaller tasks, opportunities for scal-

ing to more processors increase. GPUs execute especially fine grained work loads. In GPU MD applications, one thread typically acts on a single particle.[3,5,8,11,12] Such a decomposition is not directly applicable to traditional MC because each trial move depends on the state of the neighboring particles.

In MPMC, we utilize the *cell list* data structure for parallel decomposition, as well as overlap checks in the case of hard particles. A checkerboard decomposition permits many cells to be updated independently.[17] Although similar to applying trial moves to particles in a particular sequential order, checkerboard decomposition differs from the serial algorithm in one key way. Particle positions, not labels (or indices), determine the order of updates, so the order will change as particles migrate. Consequently, careless choices can lead to erroneous simulations. We prove that our implementation of MPMC obeys detailed balance to ensure that no incorrect choices are made in its design.

2.1 Checkerboard decomposition

The checkerboard domain decomposition scheme[17] divides the simulation volume into sets of square (cubic) cells (see Figure 2). Checkerboarding maps well to MC simulations because it allows parallel updates of each set, comprising one quarter (one eighth in three dimensions) of the simulation volume. The x and y coordinates of the cell (and z in three dimensions) determine its checkerboard set $Q \in \{a,b,c,d,\ldots\}$:

$$Q = \begin{cases} a & \text{if } (x \in \text{Even}) \quad \text{and} \quad (y \in \text{Even}), \\ b & \text{if } (x \in \text{Odd}) \quad \text{and} \quad (y \in \text{Even}), \\ c & \text{if } (x \in \text{Even}) \quad \text{and} \quad (y \in \text{Odd}), \\ d & \text{if } (x \in \text{Odd}) \quad \text{and} \quad (y \in \text{Odd}), \\ \ldots & \ldots, \end{cases} \tag{1}$$

where $a,b,\ldots$ indicate labels of checkerboard sets.

The width of the cell w must be chosen greater than the diameter of the disk σ (generally, the pair interaction cutoff). At the minimum $w = \sigma$, two particles separated by one cell can move without interacting (see Figure 2(c)). Thus, the moves available to particles in a cell are independent from those in other cells of the same checkerboard set.

Algorithm 1 Monte Carlo sweep

```
1:  C ← {a,b,c,d,...}
2:  rng.shuffle(C)
3:  for Q ∈ C do                          ▷ Loop over sub-sweeps
4:      for c ∈ cells(Q), in parallel do  ▷ Loop over cells
5:          rng.shuffle(c.particles)
6:          for s ∈ [0...n_M) do
7:              p ← c.particles[mod(s, len(c.particles))]
8:              Generate trial move
9:              if p remains in cell and move accepted then
10:                 Move p
11:             end if
12:         end for
13:     end for
14: end for
15: d ← rng.uniform(0, w/2)
16: f⃗ ← rng.choose(−x, +x, −y, +y, ...)
17: shift_cells(f⃗, d)
```

Most previous parallel MC simulations with mobile particles use stripe domain decomposition,[21–23] a one-dimensional version of the checkerboard decomposition. The number of stripes and therefore the number of trial moves that can be conducted in parallel is low. This means stripe decomposition is not efficient for parallelization on more than a few cores.

2.2 Sweep structure

Algorithm 1 outlines the structure of MPMC. It splits each sweep over cells into sub-sweeps (four in two dimensions, eight in three dimensions), one handling each checkerboard set. Line 2 shuffles the order of checkerboard sets using Fisher-Yates[30] to guarantee a random permutation. During a sub-sweep, the algorithm concurrently processes all of the cells in the active set (line 4).

Each concurrent cell update shuffles and then loops over n_M trial moves (lines 5,6). Line 7 selects the particle from the cell, repeating from the start of the list when $n_M > n$, where n is the number of particles in the cell. Fixing the number of moves in all cells to the same number distributes computational effort most evenly among GPU cores.[31]

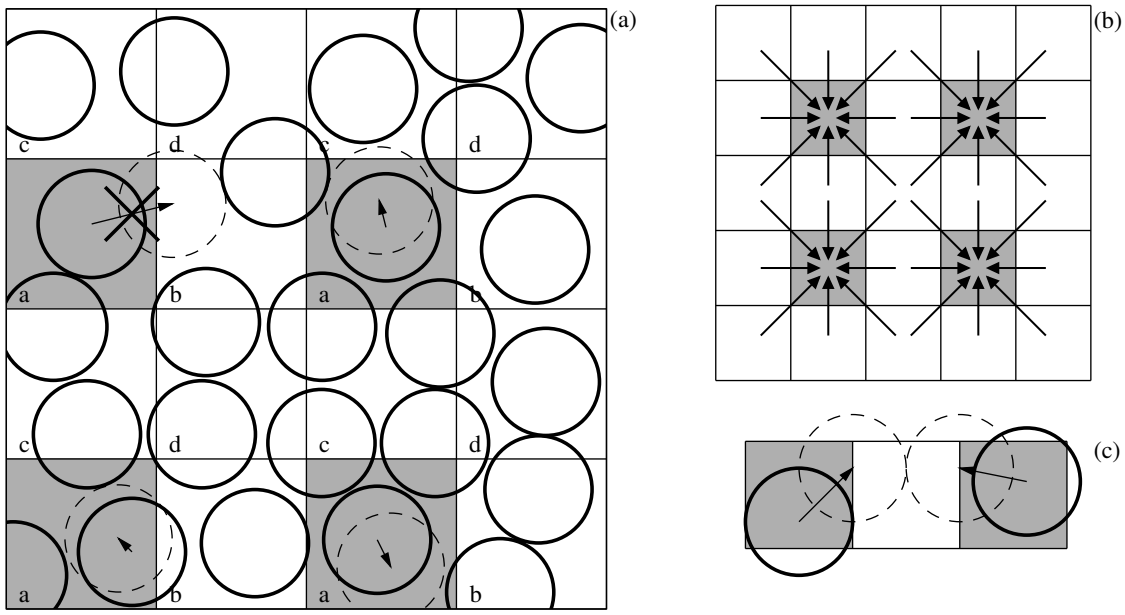

Figure 2: (a) In massively parallel Monte Carlo (MPMC), trial moves are concurrently applied to particles in a subset of the cells. Moves that leave the cell are rejected. (b) Selected cells are separated by one row or one column of inactive cells. During the evaluation of the acceptance criterion, each active cell reads the particles in the eight neighboring inactive cells. (c) Simultaneous trial moves do not interact when the cell width is greater than the interaction range σ.

Line 8 generates a trial move for each particle. Line 9 accepts the move if it passes the normal Metropolis acceptance criterion[32] and the particle center remains in the cell.[21] Lines 15-17 maintain ergodicity by performing a cell shift, which redraws the cell boundaries in a randomly chosen location.

2.3 Detailed balance

A MC simulation obeys detailed balance if, for every internal process evolving the system there exists a reverse process occurring at the same rate. This ensures that a sequence of configurations converges to the correct equilibrium distribution, regardless of the initial condition. Markov-chain MC generates a sequence of configurations where the probability $x_j(t+1)$ of observing the system in state j at step $t+1$ is determined only by the previous state i at step t. This can be expressed by

$$x_j(t+1) = x_i(t)P_{ij}, \tag{2}$$

where $x(t) = \{x_1(t), x_2(t), ..., x_n(t)\}$ is the probability distribution at step t. The elements P_{ij} of the transition matrix represent the probabilities that the system will transition from state i to state j. If there exists an equilibrium distribution of states x^* for which $x^* = x^*P$, then $x(t)$ is guaranteed to converge to x^* as $t \to \infty$ when P satisfies detailed balance:

$$x_i^* P_{ij} = x_j^* P_{ji}. \tag{3}$$

Although detailed balance of a Markov chain is a sufficient condition to ensure convergence, it is not necessary. Manousiouthakis and Deem show that an irreducible transition matrix that enforces regular sampling ($\exists m : (P^m)_{ij} > 0 \, \forall i, j$) and obeys balance ($x^* = x^*P$) is both necessary and sufficient for convergence to the correct equilibrium distribution.[33] In this work we choose to enforce detailed balance for reasons discussed later in the paper.

The MPMC algorithm constructs a Markov chain and obeys detailed balance on the level of a MC sweep. This follows directly from the observation that for each sweep there is exactly one inverse sweep, which can be seen as follows. Take a particular sequence of sub-sweeps and a sequence of n_M moves (either accepted or rejected) within each cell. The reverse sweep consists of the reverse sequence of sub-sweeps and the reverse se-

quence of moves within each cell, with each move following the negative of the original vector. For example, with $n_M = 6$ trial moves per cell and $n = 4$ particles in the cell, the original sequence would be $[0,1,2,3,0,1]$. There is exactly one particle shuffling, $[1,0,3,2]$, that generates the reverse sequence $[1,0,3,2,1,0]$. Since each sequence is chosen randomly from all possible permutations, the forward and reverse sequences occur with equal probability, and thus detailed balance holds.

2.4 Pitfalls leading to incorrect statistical sampling

Detailed balance is ensured when the following three steps are in place.

1. *The particle center must not leave the cell.*[21] If particles are allowed to leave their cells during a sub-sweep, the reverse sequence of moves cannot be generated and detailed balance is not ensured. When we skip this restriction in the hard disk system, it always develops order in the same orientation and at a lower than expected density.

2. *Shuffling the particles in each cell.* Particles entering a cell are added at the end of the cell list and cell lists are partially maintained during a cell shift. When we skip particle shuffling, a temporal memory of previous states builds up over many sweeps, violating the Markov property.

3. *Shuffling the checkerboard set.* Without shuffling of the checkerboard set, the reverse sweep cannot be generated, which violates the condition of detailed balance.

To increase the number of accepted moves per sweep one might be tempted to allow particles to leave the cell, compensating the violation of step (1) by ensuring that each particle moves exactly once per sweep. However, this procedure does not guarantee balance. Cell updates with moves that leave a cell in the 'middle' of the cell update (*i.e.* not the first or last successful move of the sub-sweep into a given neighboring cell) are not reversible and generate an incorrect probability distribution. When we apply this scheme in the hard disk system, the pressure is shifted slightly away from the correct value and the magnitude of shift depends on the maximum trial move distance.

The particle shuffling step (2) is often explicitly omitted in favor of sequential updating.[18,20,22,23] As a justification, these authors refer to the analysis of Manousiouthakis and Deem,[33] who showed that shuffling is not necessary for the Ising lattice model away from infinite temperature. However, their analysis cannot be transferred to systems of mobile particles if the sequence of particles is determined dynamically. Our simulations for hard disks confirm (Table 1) that skipping the particle shuffling step alters the pressure close to the melting transition. In contrast, the checkerboard set shuffling step (3) is not necessary for correct sampling.

Table 1: Simulations of $N = 256^2$ hard disks with particle (P.) and checkerboard (CB.) shuffling enabled or disabled. The comparison of equilibrium pressures P^* for runs at three different packing fractions ϕ demonstrates the necessity to shuffle particles. Checkerboard shuffling is not required to obtain correct results.

Shuffling		P^* in the hard disk system at		
P.	CB.	$\phi = 0.698$	$\phi = 0.708$	$\phi = 0.716$
Yes	Yes	9.17079(5)	9.18214(6)	9.1774(2)
Yes	No	9.1707(1)	9.1821(2)	9.1775(3)
No	Yes	9.1716(1)	9.1876(2)	9.1831(3)
No	No	9.1715(1)	9.1873(1)	9.1823(2)

3 Implementation

We implement MPMC for hard disks using the NVIDIA CUDA programming model and execute benchmarks on a GeForce GTX 680 graphics processor. CUDA is an established parallel programming language; details may be found in the CUDA programming guide,[34] text books[35–37] or in other publications, Refs.[2,3] for example. The pseudocode presented in this paper is general enough that it could be adapted to any data-parallel language (*e.g.* OpenCL or OpenMP).

3.1 Data Structures

The proper choice of data structures can make or break an implementation's performance. We keep all of the architecture details of NVIDIA GPUs in mind when designing our implementation. MD codes store particles in a flat array with N elements, and auxiliary data structures indirectly reference this list by index.[3,5,11,12] Such a data structure is not appropriate for MC as it would be expensive to rebuild the cell list every sweep.

Instead, we store the particle positions directly in the cell list. That data is a sparse flat array, $\texttt{disk}[x,y,i]$, with storage for $m \cdot m \cdot n_{\max}$ particle positions, where m is the number of cells on the side of the simulation box and $n_{\max}$ is the maximum number of particles allowed in a cell. The auxiliary array, $\texttt{n}[x,y]$ stores the number of particles in each cell, where the particles are placed in elements $i \in [0 \ldots n)$.

We minimize the number of overlap checks and maximize parallelism by setting the cell width w small, but not so small that a large fraction of moves will cross the cell boundaries. The size must also be chosen so that $n_{\max}$ is known. Figure 3 shows that the largest cell that can fit no more than four particles has a width $w < \sqrt{2}\sigma$. Furthermore, we avoid expensive boundary condition checks by choosing m as a multiple of 2 times the block size, because each thread handles every other cell. For the 32-thread blocks used here, we set $m = \lfloor L/(\sqrt{2}\sigma) \rfloor$ and then round up to the nearest multiple of 64.

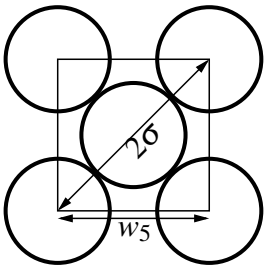

Figure 3: Four disks of diameter σ are placed on the corners of the cell and a fifth in the center. The smallest cell that can contain five disk centers has a diagonal of 2σ and an edge length of $w_5 = \sqrt{2}\sigma$. Thus, the largest cell that can contain a maximum of four disk centers has a width $w < \sqrt{2}\sigma$.

Each cell has a local coordinate system to mitigate floating point cancellation errors that would otherwise occur for large absolute coordinate values. Whenever difference vectors are computed a coordinate system transformation is needed. The components of the translation vector are either $+w$, $-w$, or 0 depending on the relative location of the neighboring cell.

3.2 Kernel

Algorithm 2 implements the MPMC sub-sweep (lines 4–13 of Algorithm 1) update in a CUDA kernel. One thread is launched for each cell in the active checkerboard set. Lines 1 and 2 compute the (x,y) index of the cell to which the thread is assigned. Rows in a thread block handle rows in the cell data. For example, threads with x ids 0,1,2,3 are assigned to cells with x coordinates 0,2,4,6 relative to some offset. Similarly, each row in a thread block is assigned to alternating y rows in the cell data.

Each thread initializes its own random number stream (line 3). We use the Saru PRNG, developed by Steve Worley,[38] to create uncorrelated random number streams from a hash of the thread index, current step index and a user chosen seed. NVIDIA's CURAND library is an alternative, but requires reading and writing a large state in each thread, which slows performance by 30%. See Ref.[39] for more details on the tradeoffs of various parallel PRNG schemes.

Lines 4–9 read the assigned cell into shared memory and shuffle the particles. Line 11 starts a loop over n_M trial moves. For each selected particle i in sequence, line 12 generates the trial move and lines 13–23 check for any overlaps with particles in the neighboring cells. Lines 24-28 update the particle to its new position if the move generates no overlaps and remains in the cell. Lines 39–32 wrap i back to the start of the cell when the end is reached.

When checking overlaps, each thread reads the eight neighboring cells. Typical GPU kernels with this memory access pattern use shared memory as a managed cache to avoid multiple reads from the same cell. In our case, the data size per cell is large and would occupy a substantial fraction of the available shared memory, limiting parallelism. Instead, we read only the current cell into shared

Algorithm 2 Subsweep GPU kernel

Require: `gdim` is $(m/\texttt{bdim.x}/2, m/\texttt{bdim.y}/2)$
Require: $(\texttt{off.x}, \texttt{off.y})$ is the offset to the lower-leftmost active cell in the sub sweep
Require: `seed` is a random number seed chosen by the user and fixed for the duration of a run
Require: `sweep` is the index of the current sweep
Require: $\vec{D}_{\text{sh}}[p]$ is stored in shared memory such that each thread indexes unique elements in memory for $p \in [0 \ldots n_{\max})$

1: $x \leftarrow 2\,(\texttt{bidx.x} \cdot \texttt{bdim.x} + \texttt{tidx.x}) + \texttt{off.x}$
2: $y \leftarrow 2\,(\texttt{bidx.y} \cdot \texttt{bdim.y} + \texttt{tidx.y}) + \texttt{off.y}$
3: $\texttt{rng} \leftarrow \textbf{Saru}(mx+y, \texttt{step}, \texttt{seed})$
4: $n \Leftarrow \texttt{n}[x,y]$
5: **if** $n == 0$ **then**
6: **return**
7: **end if**
8: $\vec{D}_{\text{sh}}[i] \Leftarrow \texttt{disk}[x,y,i]\ \forall i \in [0 \ldots n_{\max})$
9: **rng.shuffle**$(\vec{D}_{\text{sh}}[0 \ldots n])$
10: $i \leftarrow 0$
11: **for** $s \in [0 \ldots n_M)$ **do**
12: $\vec{D}_{\text{move}} \leftarrow \vec{D}_{\text{sh}}[i] + \textbf{rng.inCircle}(d)$
13: $\texttt{overlap} \leftarrow \textbf{False}$
14: **for** $(x_{\text{neigh}}, y_{\text{neigh}}) \in$ neighborhood of cell (x,y) **do**
15: $\vec{s} \leftarrow$vector pointing to current neighbor
16: **for** $j \in [0 \ldots n_{\max})$ **do**
17: **continue** when $(x_{\text{neigh}}, y_{\text{neigh}}) = (x,y) \wedge i = j$
18: $\vec{D} \Leftarrow \texttt{disk}[x_{\text{neigh}}, y_{\text{neigh}}, j]$
19: **if** $\left|\vec{D}_{\text{move}} - (\vec{D} + \vec{s})\right| < \sigma$ **then**
20: $\texttt{overlap} \leftarrow \textbf{True}$
21: **end if**
22: **end for**
23: **end for**
24: **if** $\neg\texttt{overlap}$ **then**
25: **if** $\vec{D}_{\text{move}} \in$ cell (x,y) **then**
26: $\vec{D}_{\text{sh}}[i] \leftarrow \vec{D}_{\text{move}}$
27: **end if**
28: **end if**
29: $i \leftarrow i+1$
30: **if** $i \geq n$ **then**
31: $i \leftarrow 0$
32: **end if**
33: **end for**
34: $\vec{D}_{\text{sh}}[i] \Rightarrow \texttt{disk}[x,y,i]\ \forall i \in [0 \ldots n_{\max})$

Algorithm 3 Index cell data

1: **procedure** CELL_INDEX(x, y, i)
2: **if** $x \in$ Odd **then**
3: $q \leftarrow (x+m)/2$
4: **else**
5: $q \leftarrow x/2$
6: **end if**
7: **return** $(i \cdot m + y) \cdot m + q$
8: **end procedure**

memory (line 8) and use hardware cached reads for the neighbor accesses (line 18).

A 128-byte wide cache line in GTX 680 fits four full cells in a row. However, a row-major assignment of $[x,y,i]$ to a linear index is not ideal. Only one particle would be read at a time from alternating cells in a row, using 8 out of the 128 bytes in a delivered cache line. A carefully chosen mapping from $[x,y,i]$ indices to linear memory addresses leads to full utilization. Algorithm 3 implements that mapping. First, i is the slowest index so that memory instructions in loops over i in the kernel read contiguous data. The next fastest index is y which is handled in the traditional manner. The x index is the fastest, but it is rearranged so that all the odd, and similarly even, x values are contiguous in the linear space. We achieve further performance improvements by using texture reads (`tex1Dfetch`) in place of all global memory loads. On GTX 680, the texture cache provides more throughput than L1 for read-only data.

Thanks to these efforts, we achieve excellent utilization of the available memory bandwidth. Benchmarks with the NVIDIA visual profiler show a sustained bandwidth of $\sim$ 200 GB/s out of the texture cache. Achieving high occupancy and limiting divergence are just as important. For example, selecting the 16k/48k (L1/shared) mode increases occupancy and boosts performance by 50% compared to the 48k/16k mode. The loop on line 16 goes from 0 to $n_{\max}$ to boost performance by reducing divergent branches compared to looping over the number of particles currently in the cell. We set the values of the empty particle slots to $(-10,-10)$ so that they do not result in false overlaps. Early exit conditions upon finding the first overlap (not shown) cause additional divergent branches, but removing these checks reduces

Algorithm 4 Cell shift GPU kernel

Require: `gdim` is $(m/\texttt{bdim.x}, m/\texttt{bdim.y})$
Require: $\vec{D}_{\text{sh}}[p]$ is stored in shared memory such that each thread indexes unique elements in memory for $p \in [0 \ldots n_{\max})$

```
1:  procedure SHIFT_CELLS(f⃗, d)
2:      x ← (bidx.x · bdim.x)
3:      y ← (bidx.y · bdim.y)
4:      n_current ⇐ n[x, y]
5:      D⃗_sh[i] ← (−10, −10) ∀i ∈ [0 ... n_max)
6:      n_new ← 0
7:      for i ∈ [0 ... n_current) do
8:          D⃗ ⇐ disk[x, y, i]
9:          D⃗ ← D⃗ − f⃗ · d
10:         if D.x > 0 ∧ D.y > 0 ∧ D.x ≤ w ∧ D.y ≤ w then
11:             D⃗_sh[n_new] ← D⃗
12:             n_new ← n_new + 1
13:         end if
14:     end for
15:     (x_neigh, y_neigh) ← cell in direction of f⃗
16:     s⃗ ← vector pointing to neighbor
17:     n_neigh ⇐ n[x_neigh, y_neigh]
18:     for i ∈ [0 ... n_neigh) do
19:         D⃗ ⇐ disk[x_neigh, y_neigh, i]
20:         D⃗ ← D⃗ − f⃗ · d
21:         if D.x > 0 ∧ D.y > 0 ∧ D.x ≤ w ∧ D.y ≤ w then
22:             ▷ Particle stays in neighbor cell, do nothing
23:         else
24:             D⃗ ← D⃗ + s⃗
25:             D⃗_sh[n_new] ← D⃗
26:             n_new ← n_new + 1
27:         end if
28:     end for
29:     D⃗_sh[i] ⇒ disk_dbl[x, y, i] ∀i ∈ [0 ... n_max)
30:     n_new ⇒ n_dbl[x, y]
31: end procedure
```

performance due to the increase in computations and memory accesses.

Kernel performance varies with block size.[3] Short benchmarks show that (32, 4) is the fastest, and we use it for all production runs. It outperforms the slowest by 55%, demonstrating the importance of performing this test.

Algorithm 4 implements the cell shift step on the GPU by equivalently translating the particles in the opposite direction. One thread per cell gathers all of the particles that belong in the new cell and builds the list in shared memory. It then writes out the cell to a separate memory area, `disk_dbl` and `n_dbl`, so that other running threads do not read updated data. After the kernel completes, the double buffered data structures are swapped. Since the shift direction is one of either $+x$, $-x$, $+y$, or $-y$, only two old cells contribute particles to the new cell: the cell with the same index, and one neighbor. Lines 7–14 loop over the current cell, read in each particle, shift the cell, and if that particle is still within the cell boundaries it is added to the new list. Lines 18–28 perform the same operations on the neighbor cell, with the addition of a coordinate system transformation. These rely on the following logic: if the particle left its host cell, it must have entered this cell. In this manner, each particle is checked for inclusion by two threads and at two separate points in the code.

The floating point operations by both of these threads *must* be identical. Consider if line 20 were to perform the coordinate system translation $\vec{D} - \vec{f} \cdot d + \vec{s}$ and then check for particles that enter the current cell. Floating point round-off errors may result in both this check and the corresponding check on line 10 to fail, losing the particle. In a test simulation configured with 1024^2 particles, several hundred were lost after 10^6 sweeps. When implemented as shown in Algorithm 4, no particles are lost even after 10^9 sweeps.

3.3 Parameter tuning

The maximum move radius d and the number of trial moves performed in each cell update n_M are free parameters. At fixed n_M, we test $d \in [0.06, 0.08, \ldots 0.20]$ and at fixed d, we test $n_M \in [1 \ldots 8]$. Each test measures the autocorrelation of the average orientational order parameter[40] over a

long run of 10^9 sweeps for $N = 512^2$. We find that $d = 0.16$ and $n_M = 4$ minimizes the autocorrelation time τ when measured in wall clock seconds.

4 Results

The hard disk system is a standard model system in statistical mechanics and the one which was originally used to pioneer the Monte Carlo computer simulation method.[32] Its phase behavior is completely determined by the equation of state, which is the relation between internal pressure P and packing fraction $\phi = \rho\pi(\sigma/2)^2$. Here, $\rho = N/V$ is density and V the volume of the simulation box. At packing fractions between $\phi = 0.7$ to 0.72 the system undergoes a first-order phase transition from the liquid phase to the hexatic phase, followed by a continuous transition to the solid phase.[29] In their paper, Bernard and Krauth showed with an event-chain simulation method[16] that only very large simulations ($N > 256^2$) have minimal finite size effects, and long equilibration times are necessary. This demonstrates the need for high performance MC code and explains why previous simulation studies of the hard disk system[41–46] have been unable to provide conclusive evidence for a first-order phase transition or the existence of an intermediate hexatic phase.

Phase transitions are extremely sensitive to the slightest programming error or inadvertent correlation of trial moves due to the appearance of (quasi-)long-range spatial correlations and long equilibration times. Structural fluctuations are more important in two dimensions than in three dimensions, and they are particularly large for the hard disk system. These properties, together with the availability of high-precision serial data to compare with, make hard disks a good system to test our algorithm.

In principal, computing the pressure for hard disks is simple: Estimate the radial distribution function $g(r)$ in the limit as r approaches the disk diameter σ from the right, and calculate[32]

$$P^* = \frac{P\sigma^2}{k_\mathrm{B}T} = \sigma^2\rho\left(1 + \frac{\pi}{2}\sigma^2\rho \lim_{r\to\sigma+} g(r)\right), \quad (4)$$

where we introduce the dimensionless pressure P^*. In practice, obtaining an unbiased estimate requires special care. We estimate pressure using the following procedure. For each sampled configuration, a histogram of particle pair distances is computed over all N particles. $n[r_i]$ counts the number of particle pairs between r_i and $r_i + \delta r$. The pair distribution function $g(r)$ is evaluated by the equation

$$g(R_i) = \frac{n[r_i]}{N\rho\,\delta A} = \frac{n[r_i]}{N\rho\,2\pi R_i\,\delta r} \quad (5)$$

at the sampling points $r = R_i$, where

$$R_i = \frac{2}{3}\frac{{r_{i+1}}^3 - {r_i}^3}{{r_{i+1}}^2 - {r_i}^2}. \quad (6)$$

The formula for R_i is derived assuming a linear dependence of $n(r)$ with r, which we observe to be valid close to σ. The minimum position R_0 is greater than σ, so direct evaluation of the limit is not possible. We fit $g(R_i)$ to a polynomial of degree d in the range $r \in (\sigma, \sigma + c]$ and extrapolate to $r = \sigma$. Extensive testing with a model distribution ($n(r) = 30e^{-30r}$) tunes the parameters to ensure that there is no systematic bias. We choose parameters in the middle of the flat region with systematic errors less than 10^{-5}. They are $\delta r = 10^{-4}\sigma$, $c = 0.02\sigma$, and $d = 5$.

Table 2 shows the average P^* data obtained by simulations with MPMC. Independent runs of $N = 256^2$, $N = 512^2$, and $N = 1024^2$ particles are performed at packing fractions between $\phi = 0.698$ and $\phi = 0.718$, which comprises the transformation from liquid to solid. Each run starts with a randomly generated configuration at low density and is quickly compressed to the target. The run then continues at constant density for 10^9 sweeps, which only takes 4 days for $N = 256^2$ (15 days for $N = 512^2$) to complete on a Tesla M2070 GPU. The pressure is averaged every 200 sweeps in each run after an equilibration period of $3 \cdot 10^8$ sweeps.

Our data confirms the equation of state reported in Ref.[29] All values overlap within error bars. We do not analyze positional order or orientation order in the dense phase emerging from the phase transition. Such an analysis would be necessary to distinguish a hexatic phase from a solid phase and is left for a separate work.[40] We refer to Ref. 41 for an in-depth comparison of the phase diagram of hard disks obtained with various algorithms, including MPMC.

Table 2: This table shows data for the equation of state $P^*(\phi)$ over the range where the liquid transforms into the solid. It includes runs by MPMC (this work) and by Bernard and Krauth (BK) with serial event chain simulation.[29] Error bars are shown at two standard errors of the mean, $\sigma = 2(\langle[P^* - \langle P^*\rangle]^2\rangle/N_{\text{samples}})^{1/2}$, where the number in parentheses is the error in the last digit shown. Our data is averaged over 8 independent runs of 10^9 sweeps (64 runs for $\phi = 0.718$, $N = 256^2$). See Ref.[29] for a description of the BK data averaging scheme. Differences in the pressures are shown with propagated error bars $(\sigma^2_{\text{MPMC}} + \sigma^2_{\text{BK}})^{1/2}$.

System size	Method	Dimensionless pressure P^* in the hard disk system at packing fraction $\phi = 0.698$	$\phi = 0.702$	$\phi = 0.706$	$\phi = 0.710$	$\phi = 0.714$	$\phi = 0.718$
$N = 256^2$	MPMC	9.1709(1)	9.1920(2)	9.1854(1)	9.1792(1)	9.1758(1)	9.187(1)
	BK	9.1708(4)	9.1924(4)	9.1858(5)	9.1790(4)	9.1758(5)	9.186(1)
	Difference	0.0000(5)	0.0004(5)	0.0004(5)	0.0002(4)	0.0000(6)	0.001(1)
$N = 512^2$	MPMC	9.1699(5)	9.1900(2)	9.1861(1)	9.1828(1)	9.1800(1)	9.1930(4)
	BK	9.1700(2)	9.1899(6)	9.1856(6)	9.1821(5)	9.1803(4)	9.1937(2)
	Difference	0.0001(5)	0.0001(6)	0.0004(6)	0.0007(5)	0.0003(4)	0.0006(5)
$N = 1024^2$	MPMC	9.1694(1)	9.187(1)	9.1858(3)	9.1843(5)	9.1819(7)	9.21(1)
	BK	9.1693(1)	9.1880(2)	9.1855(2)	9.1843(2)	9.1822(2)	9.1949(3)
	Difference	0.0001(2)	0.001(1)	0.0002(4)	0.0000(5)	0.0003(7)	0.01(1)

5 Performance

We test the performance of the MPMC hard disk code using CUDA on a single GTX 680 GPU (Kepler). For comparison, we also implement the MPMC algorithm on the CPU and run it on our cluster nodes. The CPU implementation uses OpenMP to parallelize across all cores on a node. Each thread processes a single horizontal strip of the simulation domain and the innermost loop follows Algorithm 2. We make no attempt to parallelize across multiple GPUs or multiple CPU nodes using MPI. All tests are performed using single precision floating point format to store particle coordinates. Table 3 lists complete specifications of the hardware and software configurations of our test machines.

5.1 Scaling with number of particles

We perform benchmarks at a fixed packing fraction $\phi = 0.698$ and analyze the performance scaling with N. Figure 4(a) plots the results and Table 4 lists selected numerical values.

The algorithm has a running time $t \in O(N)$, so the number of trial moves per unit time, η, should be constant under ideal circumstances. In practice, there are overheads that cause deviation from constant efficiency. Figure 4(b) collapses the individual η plots to a relative efficiency metric for each separate configuration. When running on a single CPU core, efficiency is flat with only a slight downward drift for large N. This is likely because the largest systems no longer fit in on-chip cache. The 8-core benchmarks show the same behavior at large N, although it starts a factor of 2 higher because the same data is now split over 2 chips' caches. The 8-core results also have a slight dip for $N < 10^5$ due to the overhead of managing worker threads. On the GPU, the kernel launch overhead is significant enough that for N less than 10^5, efficiency is less than 70%.

Despite this inefficiency, the GPU still outperforms the single CPU runs by a factor of 82 for systems as small as $N = 253^2$. At peak efficiency, the GPU speedup over a single core is a factor of 95. We prefer thinking about speedups compared to a single core, but recognize that there are other ways of evaluating it. On a per socket basis, a single GPU is 24x faster than a quad core E5540. On a per-node basis, it is still an order of magnitude (12x) faster. In terms of aggregate performance per price, an 8-core node with 8 externally attached GPUs (the configuration we use) is

Table 3: Benchmark hardware and software configurations.

	GTX 680	E5540
Hardware	custom built	HP DL2x170h
Mainboard	ASUS Sabertooth	
Chipset	AMD 990FX	Intel 5520
CPU	AMD Athlon II X4 630	2x Intel Xeon E5540
CPU clock	2.8 GHz	2.53GHz
RAM	4GB DDR3	24GB DDR3
RAM clock	1333 MHz	1333 MHz
GPU	GeForce GTX 680	
Core clock	1006 MHz	
Memory clock	6008 MHz	
DRAM	2 GB GDDR5	
OS	Gentoo	RHEL 6
Architecture	x86_64	x86_64
CPU compiler	GCC 4.5.3	GCC 4.7.0
flags	-O3 -funroll-loops	-O3 -funroll-loops
GPU compiler	CUDA 4.2	
flags	*default*	
GPU driver	295.59	

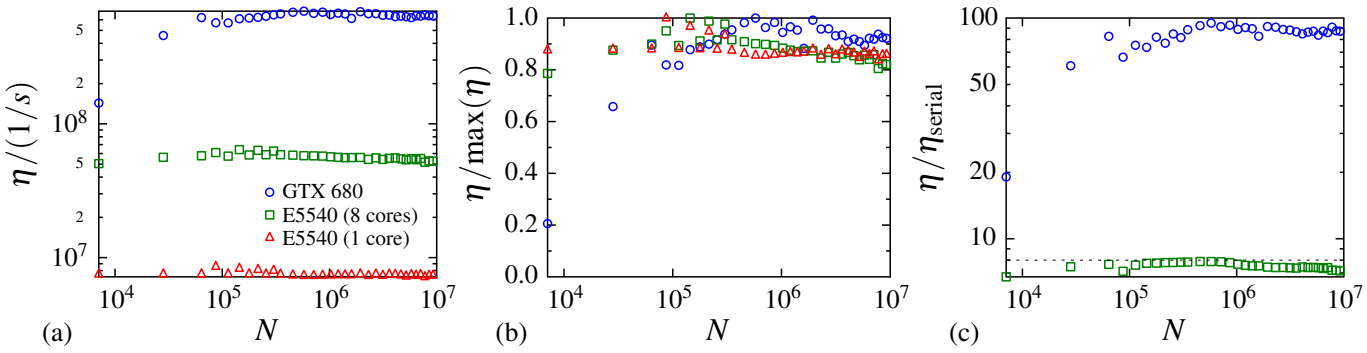

Figure 4: Benchmarks are performed with a varying number of disks, N, at a packing fraction $\phi = 0.698$. Each benchmark runs 100 sweeps to warm up and then measures the time it takes to run another 100 sweeps as well as the the number of attempted trial moves to compute $\eta = N_{\text{moves}}/t$. 11 separate runs are performed and the median result is plotted here. In all cases, the error bars (one standard deviation) are smaller than the symbol size. Subfigure (a) plots the efficiency of the computation η vs. N on various hardware configurations. Subfigure (b) plots η normalized by the maximum obtained on each hardware configuration. Subfigure (c) plots the speedup obtained over a serial execution. The dashed line marks a speedup of 8.

Table 4: Numerical values for selected benchmarks from Figure 4.

		253^2	760^2	1520^2	3040^2
GTX 680	η	$6.23 \cdot 10^8$	$6.95 \cdot 10^8$	$6.66 \cdot 10^8$	$6.41 \cdot 10^8$
	$\eta/\max(\eta)$	0.895	1.00	0.96	0.920
	$\eta/\eta_{\text{serial}}$	82.6	94.8	90.9	87.2
E5540	η	$5.77 \cdot 10^7$	$5.78 \cdot 10^7$	$5.42 \cdot 10^7$	$5.26 \cdot 10^7$
(8 cores)	$\eta/\max(\eta)$	0.900	0.900	0.860	0.820
	$\eta/\eta_{\text{serial}}$	7.65	7.88	7.39	7.16
E5540	η	$7.54 \cdot 10^6$	$7.33 \cdot 10^6$	$7.33 \cdot 10^6$	$7.35 \cdot 10^6$
(1 core)	$\eta/\max(\eta)$	0.880	0.855	0.855	0.858

95x faster than just the host, but only costs 5.5x as much for a benefit of 17x more sweeps/unit time given a fixed budget. We do not have power monitoring equipment on our cluster, so we are unable to provide actual measurements of power savings. Instead, we obtain an estimate using the manufacturer's TDP specifications, which report 80W for the E5540 and 195W for the GTX 680. Based on these numbers and the per-socket speedup, a CPU simulation would use 9.8 times more energy than if it were run for the same number of sweeps on the GPU.

5.2 Limitations

The maximum number of parallel threads depends on the cell size, interaction range, and particle shape. It is approximately $N/8$ for hard disks at high density. Hard spheres decrease the maximum to $N/16$. Expanding the interaction range to a truncated and shifted Lennard-Jones potential with $r_{\text{cut}} = 2.5\sigma$ drops the maximum to $N/144$. GPUs operate at peak efficiency only when running more than 10000 threads, establishing minimum practical system sizes of 80 thousand, 160 thousand, and 1.44 million particles for hard disks, hard spheres, and Lennard-Jones beads, respectively. Depending on the desired application, these sizes may be unnecessary or prohibitively large. A moderately threaded CPU implementation does not suffer from this problem.

6 Conclusions

Efficient parallel algorithms are essential for the application of Monte Carlo particle simulations on current and future computer hardware. Building on prior works utilizing checkerboard domain decomposition, we detailed an algorithm for massively parallel MC and implemented it on the GPU and the CPU. The GPU speedup that we obtain is comparable to what has been achieved in MD simulations. Our findings demonstrate that GPUs are well-suited for running MC simulations when the number of particles is high enough. While the small system size overhead is the only restriction of our method, it can be significant if one is interested in the behavior of systems of that size.

During the course of this work we learned that it is surprisingly difficult to implement MC in a parallel environment. Every slight violation of the balance conditions leads to incorrect sampling, and with parallel update moves it is not simple to determine which schemes do not obey balance. Sometimes the effect on our simulations was so small that it would be easy to miss in a typical complex practical application. It is important to test any new parallel algorithm thoroughly in a situation where reliable results are available from serial simulations. Hard disks are such a system. We computed to high precision the hard disk equation of state over the density range where the liquid transforms into the solid. Our results agree perfectly with serial event-chain Monte Carlo simulations and are compatible with the presence of a first-order liquid-hexatic phase transition[40] – a finding that is scientifically significant by itself.

7 Acknowledgements

We thank Werner Krauth and Etienne Bernard for discussions during the final stages of this work. J.A.A., M.E. and S.C.G acknowledge support by the Assistant Secretary of Defense for Research and Engineering, U.S. Department of Defense [DOD/ASD(R&E)](N00244-09-1-0062). Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the DOD/ASD(R&E). E.J. and S.C.G. received support from the James S. McDonnell Foundation 21st Century Science Research Award/Studying Complex Systems, grant no. 220020139, and E.J. acknowledges support from the National Defense Science and Engineering Graduate (NDSEG) Fellowship, 32 CFR 168a. E.J., T.L.G., and S.C.G. acknowledge support from the National Science Foundation under Award No. CHE 0624807. Simulations were performed on a GPU cluster hosted by the University of Michigan's Center for Advanced Computing.

References

(1) Asanovic, K.; Bodik, R.; Catanzaro, B. C.; Gebis, J. J.; Husbands, P.; Keutzer, K.; Patterson, D. A.; Plishker, W. L.; Shalf, J.; Williams, S. W.; Yelick, K. A. *The landscape of parallel computing research: a view from Berkeley*; 2006.

(2) Stone, J. E.; Hardy, D. J.; Ufimtsev, I. S.; Schulten, K. *Journal of Molecular Graphics & Modelling* **2010**, *29*, 116–25.

(3) Anderson, J. A.; Lorenz, C. D.; Travesset, A. *Journal of Computational Physics* **2008**, *227*, 5342–5359.

(4) HOOMD-blue. http://codeblue.umich.edu/hoomd-blue, 2012; http://codeblue.umich.edu/hoomd-blue.

(5) Brown, W. M.; Wang, P.; Plimpton, S. J.; Tharrington, A. N. *Computer Physics Communications* **2011**, *182*, 898–911.

(6) Götz, A. W.; Williamson, M. J.; Xu, D.; Poole, D.; Le Grand, S.; Walker, R. C. *Journal of Chemical Theory and Computation* **2012**, *8*, 1542–1555.

(7) Grand, S. L.; Götz, A. W.; Walker, R. C. *Computer Physics Communications* **2012**,

(8) Stone, J. E.; Phillips, J. C.; Freddolino, P. L.; Hardy, D. J.; Trabuco, L. G.; Schulten, K. *Journal of Computational Chemistry* **2007**, *28*, 2618–2640.

(9) Eastman, P.; Pande, V. S. *Journal of Computational Chemistry* **2010**, *31*, 1268–72.

(10) Ganesan, N.; Bauer, B. A.; Lucas, T. R.; Patel, S.; Taufer, M. *Journal of Computational Chemistry* **2011**, *32*, 2958–2973.

(11) Colberg, P. H.; Höfling, F. *Computer Physics Communications* **2011**, *182*, 1120–1129.

(12) Rapaport, D. C. *Computer Physics Communications* **2011**, *182*, 926–934.

(13) Swendsen, R.; Wang, J.-s. *Physical Review Letters* **1987**, *58*, 86–88.

(14) Liu, J.; Luijten, E. *Physical Review Letters* **2004**, *92*, 1–4.

(15) Whitelam, S.; Geissler, P. L. *Journal of Chemical Physics* **2007**, *127*, 154101.

(16) Bernard, E.; Krauth, W.; Wilson, D. *Physical Review E* **2009**, *80*, 5–9.

(17) Pawley, G.; Bowler, K.; Kenway, R.; Wallace, D. *Computer Physics Communications* **1985**, *37*, 251–260.

(18) Ren, R.; Orkoulas, G. *Journal of Chemical Physics* **2006**, *124*, 64109.

(19) Preis, T.; Virnau, P.; Paul, W.; Schneider, J. J. *Journal of Computational Physics* **2009**, *228*, 4468–4477.

(20) Levy, T.; Cohen, G.; Rabani, E. *Journal of Chemical Theory and Computation* **2010**, *6*, 3293–3301.

(21) Uhlherr, A.; Leak, S. J.; Adam, N. E.; Nyberg, P. E.; Doxastakis, M.; Mavrantzas, V. G.; Theodorou, D. N. *Computer Physics Communications* **2002**, *144*, 1–22.

(22) Ren, R.; Orkoulas, G. *Journal of Chemical Physics* **2007**, *126*, 211102.

(23) O'Keeffe, C. J.; Orkoulas, G. *Journal of Chemical Physics* **2009**, *130*, 134109.

(24) Sadigh, B.; Erhart, P.; Stukowski, A.; Caro, A.; Martinez, E.; Zepeda-Ruiz, L. *Physical Review B* **2012**, *85*, 1–11.

(25) Lubachevsky, B. D. *Complex Systems* **1987**, *1*, 1099–1123.

(26) Korniss, G.; Novotny, M.; Rikvold, P. *Journal of Computational Physics* **1999**, *153*, 488–508.

(27) Esselink, K.; Loyens, L.; Smit, B. *Physical Review E* **1995**, *51*, 1560–1568.

(28) Loyens, L.; Smit, B.; Esselink, K. *Molecular Physics* **1995**, *86*, 171–183.

(29) Bernard, E.; Krauth, W. *Physical Review Letters* **2011**, *107*, 1–4.

(30) Durstenfeld, R. *Communications of the ACM* **1964**, *7*, 420.

(31) Krauth, W. Personal communication, 2012.

(32) Metropolis, N.; Rosenbluth, A. W.; Rosenbluth, M. N.; Teller, A. H.; Teller, E. *Journal of Chemical Physics* **1953**, *21*, 1087.

(33) Manousiouthakis, V. I.; Deem, M. W. *Journal of Chemical Physics* **1999**, *110*, 2753.

(34) NVIDIA, CUDA C programming guide, v4.2. 2012.

(35) Kirk, D. B.; Hwu, W.-m. W. *Programming Massively Parallel Processors: A Hands-on Approach*; Morgan Kaufmann, 2010; p 280.

(36) Farber, R. *CUDA Application Design and Development*; Morgan Kaufmann, 2011; p 336.

(37) Sanders, J. *CUDA by Example*; Addison-Wesley Professional, 2010; p 312.

(38) Worley, S. Random number generator Saru. Personal communication, 2008.

(39) Phillips, C. L.; Anderson, J. A.; Glotzer, S. C. *Journal of Computational Physics* **2011**, *230*, 7191–7201.

(40) Anderson, J. A.; Engel, M.; Glotzer, S. C.; Isobe, M.; Bernard, E. P.; Krauth, W. *submitted* **2012**,

(41) Alder, B.; Wainwright, T. *Physical Review* **1962**, *127*, 359–361.

(42) Lee, J.; Strandburg, K. *Physical Review B* **1992**, *46*, 11190–11193.

(43) Zollweg, J.; Chester, G. *Physical Review B* **1992**, *46*, 11186–11189.

(44) Weber, H.; Marx, D.; Binder, K. *Physical Review B* **1995**, *51*, 14636–14651.

(45) Jaster, A. *Physical Review E* **1999**, *59*, 2594–2602.

(46) Mak, C. *Physical Review E* **2006**, *73*, 1–4.